Efficient GPGPU-based parallel packet classification

Che-Lun Hung
Department of Computer Science and Communication Engineering
Providence University
Taichung, Taiwan
clhung@pu.edu.tw

Yaw-Ling Lin, Kuan-Ching Li
Department of Computer Science and Information Engineering
Providence University
Taichung, Taiwan
yllin@pu.edu.tw, kuancli@pu.edu.tw

Hsiao-Hsi Wang, Shih-Wei Guo
Department of Computer Science and Information Management
Providence University
Taichung, Taiwan
hhwang@pu.edu.tw, cometlcc@gmail.com

***Abstract*—With the rapid growth of network technologies, many new web services have been developed to provide various applications and computing functions. These services rely deeply on the internet. Therefore, packet classification is an important issue of network security that typically adopts a flexible packet filtering system to classify each processed packet. Traditional packet classification requires hung computing time to process large amount of internet packets. Hence, we propose a GPGPU-based parallel packet classification method to decrease the computational cost. We also evaluate the performance of the proposed method with implementation on various memory architectures of CUDA device. The experiment results demonstrate that the proposed method can achieve significant speed up over the sequential packet classification algorithms on single CPU.**

Keywords: Packet classification; GPGPU; Parallel process; Packet filtering

I. INTRODUCTION

With the rapid growth of network technologies, various platforms and web services have been developed to satisfy varied requirements on internet. Recently, cloud computing has become very popular to provide various services, such as Google App Engine, Amazon C3, Microsoft Azure, and etc. All of these services deeply rely on internet. Therefore, packet analysis is an important issue of network security of cloud platform [1]. It can control which packet data can flow to and from a network. The criteria that use when inspecting packets are based on the Layer 3 (IPv4 and IPv6) and Layer 4 (TCP, UDP, ICMP, and ICMPv6) headers. The commonly used criteria are source and destination address, source and destination port, and protocol.

Packet analysis typically relies on a packet filtering system. Filter rules are used to determine the resulting action that a packet should be dropped or passed. Filter rules are compared in sequential order, first to last. Until the packet matches a rule containing the keyword, the packet will be compared against all filter rules before the final action is taken. The last matched rule will dictate what action to take on the packet. There is an implicit pass all at the beginning of a filtering rule set meaning that if a packet does not match any filter rule the resulting action will be pass. In general, each incoming packet can be considered independently of any other packet. Although for the IP fragments, the first fragment is related to other fragments apparently but arriving without order. It can be considered as independent for classification [2].

In current network platforms, such as data center and cloud computing service, huge amount of packets are delivering at any moment. Sequential software packet classifiers often take longer to classify a packet set captured off a giga-bit network interface than it took the set to arrive, making them infeasible for real-time traffic analysis [2]. Therefore, it is an issue that analyzes a hundred of thousands of packets without affecting the network Quality-of-Service (QoS). Parallel processing can be an alternative solution in analyzing a number of packets. The performance of packet classification can be improved by using the correct hardware medium significantly.

Recently, many literatures tried to use General-Purpose Graphics Processing Unit (GPU) to solve computation intensive problem of various domains [2, 3, 4, 5]. GPGPU programming has been successfully utilized in the scientific computing domains which involve a high level of numeric computation. However, other applications also could be successfully parallelized by GPGPU. The greatest benefit is that the processing units grow from many (CPU, about 2-16) to massive (GPU, about 128-512). In 2006, NVIDIA proposed the Compute Unified Device Architecture (CUDA). CUDA uses a new computing architecture named Single

Instruction Multiple Threads (*SIMT*) [6]. This architecture allows thread to execute independent and divergent instruction streams, facilitating decision based execution which is not provide for by the more common Single Instruction Multiple Data (SIMD).

In this paper, we propose an efficient method to classify huge number of packets simultaneously by using GPGPU device. By leveraging nVidia CUDA device can achieve low cost, commodity GPU co-processors to accelerate packet filtering throughput. The packet classification could be utilized in high-resolution real-time network monitoring and long-term packet capture analysis. We also implement the proposed packet classification algorithm on a variety of memory architectures on GPU to discuss the performance of proposed method. The experiment results demonstrate that the proposed method can achieve 10X speed up over the sequential packet classification software on single CPU. It presents that GPGPU is useful for real-time traffic analysis.

The structure of this paper is as follows. Section 2 introduces the related works of packet classification. Section 3 describes the proposed method. Section 4 presents the experiment results. We conclude with section 5, providing a brief summary and conclusion.

II. RELATED WORKS

According the previous literature [1], the classification can be categorized into three types: IP routing, packet demultiplexing and packet analysis. The slight differences of these types are between target environments. IP routing is utilized to forward incoming packets through the correct interface to a destination host. Packet demultiplexing is concerned that forwarding the packets to the next hop router, or dropping the packets altogether. Packet analysis is similar in many respects to demultiplexing, and often depends on similar filtering algorithms, but may process a wider variety of packets, with a broader range of destinations. All of these three types are depending on the packet filtering. A filter is a predicate function that operates over a collection of criteria to compare each arriving packet [1, 7, 8]. Generally, a packet is classified by a filter which has the specific criteria. Filter criteria are Boolean valued comparisons, performed between values contained in discreet bit-ranges in the packet header and static protocol defined values.

The commonly-used and most reliable methods of classifying packet data are exhaustive search algorithms which compare packets against each and every filter in the filter set until a exact match is found [1, 9]. These algorithms are generally slow, and thus not very useful. Other classes of algorithms include decision tree, decomposition and tuple space approaches. Decision tree algorithms are diverse in design, but all leverage a sequential tree like traversal of a specialized data structure in order to narrow down the number of criteria against which the packet need to be compared [1, 7, 10, 11]. Most of demultiplexing and analysis filters are highly sequential approaches based on decision trees [7, 12, 13], and thus are suitable to the processing on CPUs. In contrast, decomposition algorithms can be equipped on parallel processing hardware such as FPGAs, typically splitting filter classifications into smaller sub-classifications which can be performed in parallel [1, 14, 15]. Tuple space algorithms are highly specialized, and exploit a variety of filter set properties in order to reduce processing time [1].

Most of demultiplexing algorithms adopt decision tree approaches because of their efficiency at pruning redundant computation on sequential processors. BPF is a well known algorithm that adopted Control Flow Graphs (CFGs) in an assembler style programmable pseudo-machine to improve performance on register-base processor [13]. Mach Packet Filter (MPF) and Dynamic Packet Filter (DPF) are extended from BPF. MPF was designed to extend and improve demultiplexing performance [16], while DPF focused on exploiting dynamic code generation in order to prune redundant instructions [12]. These filters led to the development of BPF+ [7], which adopted techniques such as Predicate Assertion Propagating and Partial Redundancy Elimination, in conjunction with Just-In-Time (JIT) processing and various other optimizations, to dramatically improve processing speeds. Extensible Packet Filter (xPF) [17], Fairly Fast Packet Filter (FFPF) [18] and SWIFT [19], were developed to reduce the context switching overhead, facilitate high performance demultiplexing between multiple network monitoring application, and reduce filter update latency to support real-time filter updates, respectively.

Presently, RFC algorithm [20], which is a generalization of cross-producting [21], is the fastest classification algorithm in terms of the worst-case performance. Bitmap compression has been used in IPv4 forwarding [22, 23] and IPv6 forwarding [24]. It is applied to classification to compress redundant storage in data structure [25]. However, the performance bottleneck of these methods are searching the compressed tables, and thus additional techniques have to be introduced to improve the inefficiency of calculating the number of bits set in a bitmap. Lulea [22] algorithm utilizes a summary array to pre-process the number of bits set in the bitmap, and thus it needs an extra memory access operation per trie-node to search the compressed table. The Bitmap-RFC [26] employs a built-in bit-manipulation instruction to calculate the number of bits set at runtime and apply bitmap compression to reduce its memory requirement to solve the problem of memory explosion. Thus, it is much more efficient than Lulea's in terms of time and space complexity.

However, these sequential packet classification algorithms take longer to classify a packet set captured off a giga-bit network interface than it took the set to arrive, making them infeasible for real-time traffic analysis.

Alastair *et al*., [2, 3] proposed a classification algorithm, by utilizing GPU co-processors to accelerate classification throughput and maximize processing efficiency in highly parallel execution context. They provided valuable articles for introducing the concept of parallel packet classification on CUDA and OpenCL platforms. However, these literatures are lack of the performance comparison and implementation with a variety of memory architectures of GPU. Han *et al*., [27] proposed a GPU-based IP routing approach named PacketShader. The experiment results show that GPU-based IP routing algorithm can enhance the performance over the

CPU-based IP routing approaches. These articles present an alternative of developing the parallel packet classification by leveraging GPU devices.

III. METHOD

A. *GPGPU programming*

As the GPU has become increasingly more powerful and ubiquitous, researchers have begun developing various non-graphics, or general-purpose applications [5]. Traditionally, the GPUs are organized in a streaming, data-parallel model in which the co-processors execute the same instructions on multiple data streams simultaneously. Modern GPUs include several (tens to hundreds) of each type of stream processor, both of graphical and general-purpose applications thus are faced with parallelization challenges [28].

nVidia released the Compute Unified Device Architecture (CUDA) SDK to assist developers in creating non-graphics applications that run on GPUs. A CUDA programs typically consist of a component that runs on the CPU, or host, and a smaller but computationally intensive component called the kernel that runs in parallel on the GPU [15]. Input data for the kernel must be copied to the GPU's on-board memory from CPU's main memory through the PCI-E bus prior to invoking the kernel, and output data also should be written to the GPU's memory first. All memory used by the kernel should be pre-allocated.

Kernel executes a collection of threads that computes a result for a small segment of data. To manage multiple threads, kernel is partitioned into thread blocks, with each thread block being limited to a maximum of 512 threads. The thread blocks are usually positioned within a one or two dimensional grid. Each thread can be positioned within a given block where it belongs, and this given block can be positioned within the grid. Therefore, each thread can calculate which elements of data to operate on, and which regions of memory to writhe output to by an algebraic formula. Each block is executed by a single multiprocessor, which allows all threads within the block to communicate through on-chip shared memory. The parallelism architecture of GPGPU is illustrated in Fig. 1.

B. *Memory architectures on CUDA device*

CUDA devices provide access to several memory architectures, such as global memory, constant memory, texture memory, share memory and registers, with their access latencies and limitations. The performance of device is relevant to the memory variants. Figure 2 illustrates the memory architectures of CUDA device.

Global Memory

Global memory is the biggest memory region available on CUDA devices and is capable of storing hundreds of megabytes of data. However, the access latency is highest than others. It is the critical bottleneck in kernel execution to significantly improve the throughput in data intensive application by coping data between main memory and Global memory on CUDA device. In the Fermi architecture [6], the L1 cache per SM multiprocessor is configurable to support the global memory operations. Therefore, the performance of accessing global memory has been improved.

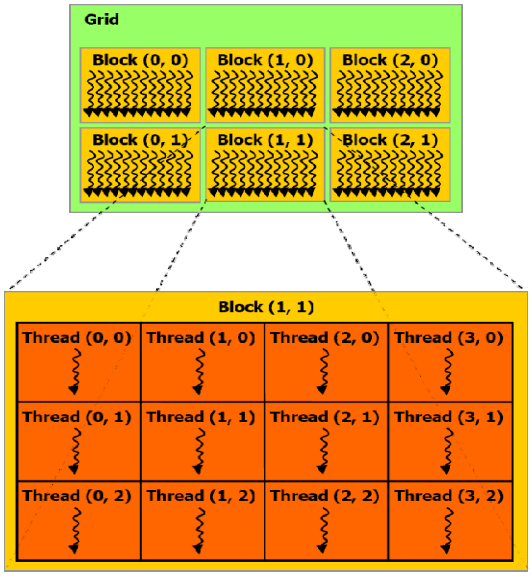


Figure 1. The parallelism architecture of GPGPU [6].

Constant Memory

Constant memory is a small read-only memory region that resides in DRAM on CUDA device. It is globally accessible memory for all threads. Since Constant memory has on-chip cache, the access latency can be reduced greatly. The cost of a cache-miss on constant memory is as a global memory access on device. On the contrary, the cost of a cache-hit is as a local register access. However, it is limited to the storage size.

Texture memory

Texture memory is a compromise between global and constant memory. Each multi-processor on the CUDA device equips a 64KB texture cache which can be bound to one or more arbitrarily sized region of global memory. Texture memory is read only as constant memory.

Registers

Each block on CUDA device equips a register file that contains registers. The register provides fast thread-local storage during kernel execution. In the Fermi architecture, each multi-processor contains 32,000 32-bit registers that are shared between all threads in the executing thread block.

Shared memory

Shared memory is block-local that facilitates cooperation between multiple thread in an executing block. Shared memory is limited to 64KB of storage per multi-processor on CUDA Fermi devices. It is a severely limited resource that is shared between multiple executing blocks on a multi-processor. The access latency of shared memory is equivalent to that of register.

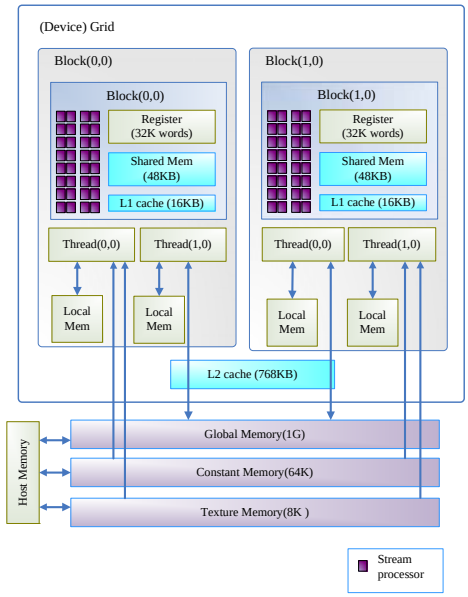


Figure 2. Memory architecture of NVIDIA GeForceGTS 450.

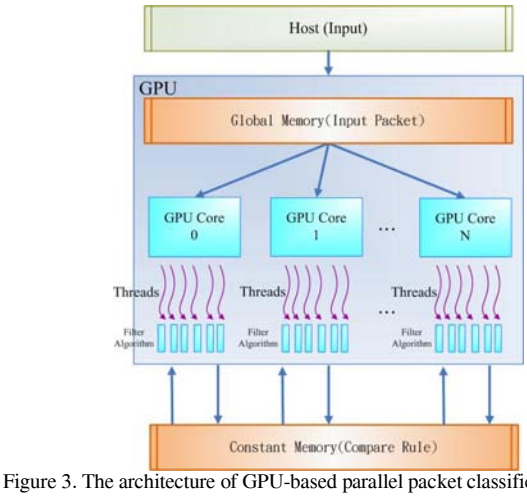


Figure 3. The architecture of GPU-based parallel packet classification.

C. *Parallel Packet Classification using GPGPU*

In CUDA devices, each physical multiprocessor contains only a single instruction register which drives eight independent processing cores simultaneously. Therefore, any divergence between thread executing on the same multiprocessor forces the instruction register to issue instruction for all thread paths sequentially whilst non-participating thread sleep [6]. The significant thread divergence can dramatically impair performance. To avoid thread divergence, each thread should process the similar length of data. In a filter set, the rules of each filter have a various number of fields. Due to this reason, we restrict that

RFC algorithm

```
Input : GpuSrcPacket , GpuDstPacket, con_eqID, con_RuleIndex, con_Ruletable, con_mask
        /* GpuSrcPacket: the source IP address of packet
           GpuDstPacket: the destination IP address of packet

           All filter rules are saved in constant memory
           RuleSize : The number of filter rules
           con_eqID: Save the Index for con_RuleIndex, and the index is transformed from IP address
           con_RuleIndex: Save the Index for con_Ruletable
           con_mask: sub-network mask if necessary
           _ _constant_ _  u_int16_t  con_eqID[RuleSize];
           _ _constant_ _  u_int16_t  con_RuleIndex [RuleSize];
           _ _constant_ _  u_int16_t  con_Ruletable [RuleSize];
           _ _constant_ _  u_int16_t  con_mask;
        */
Output : GpuPacketOut Result
Method
Begin
1.      int src,dst;
2.      int i = blockIdx.x * blockDim.x + threadIdx.x; // i is the threadID
3.      src = con_RuleIdex[con_eqID[(GpuSrcPacket + i) & con_mask]*eqIDNumber];
4.      dst = con_RuleIdex[con_eqID[(GpuDstPacket + i) & con_mask]*eqIDNumber];
5.      GpuPacketOutResult [i] = con_Ruletable[(src+dst)%256];
End
```

Figure 4. RFC algorithm on CUDA device.

BPF algorithm

```
Input : GpuSrcPacket , GpuDstPacket, con_RuleIndex, con_Ruletable, con_mask
        /* GpuSrcPacket: the source IP address of packet
           GpuDstPacket: the destination IP address of packet

           All filter rules are saved in constant memory
           RuleSize : The number of filter rules
           con_RuleIndex: Save the Index for con_Ruletable
           con_mask: sub-network mask if necessary
           _ _constant_ _  u_int16_t  con_RuleIndex [RuleSize];
           _ _constant_ _  u_int16_t  con_Ruletable [RuleSize];
           _ _constant_ _  u_int16_t  con_mask;
        */
Output : GpuPacketOut Result
Method
Begin
1.      int i = blockIdx.x * blockDim.x + threadIdx.x;
2.      for j = 0 to RuleSize − 1
3.          if ((GpuSrcPacket + i )& con_mask) == con_RuleIndex[j])
4.              GpuPacketOut Result[ i ] = con_Ruletable[j];
5.      End
End
```

Figure 5. BPF algorithm on CUDA device.

the number of fields of each rule should be the same at the same filter. Table 1 presents the revised filter rules.

Figure 3 illustrates the architecture of our GPU-based parallel packet classification method. The filter sets are stored in constant memory and register files, and the packet data is stored in global memory or texture memory. A thread process a packet data according the filter sets. Currently, we include two famous filter classification algorithms, BPF and BitMap-RFC, in the proposed method. We chose these two algorithms because they are totally different data structures. BPF is designed by decision tree and BitMap-RFC is designed by hash table. Through different data structures, it can be measured that which data structure is suitable for GPU. Figures 3 and 4 show the RFC and BPF algorithms on CUDA device. In addition, we also include Zero Copy technique [6] in the proposed method to decrease the transferring time between main memory on host and global memory on GPU. Zero copy is a feature that was added in version 2.2 of the CUDA Toolkit. It enables GPU threads to directly access host memory. Zero copy can be used in place of streams because kernel-originated data transfers automatically overlap kernel execution without the overhead of setting up and determining the optimal number of streams.

IV. EXPERIMENT

We implemented two packet classification algorithms, BPF and BitMap-RFC, on single NVIDIA GeForceGTS 450 graphics card (Fermi architecture) and installed in a PC with an Intel i3 540 3.07 GHz CPUs and 8GB DDRIII-1333 RAM running the Linux operating system. We simulated 65 million packets with the random source address, destination address, source port, destination address and protocol for the experiments. The packet filter has three rules and each rule has two fields as shown in Table 1.

A. *Perfromance Evaluation of BPFwith Various Memory Architectures*

In this experiment, we implemented BPF on CUDA with eight memory architecture combinations shown in table 2. Global memory is the biggest memory region available on CUDA devices. Constant memory and register files can access data faster than global and texture memory. However, some limitations on these two structures. First is the storage size. Constant memory is suitable for frequent access but low data update rate. The function of register on CUDA is the same as the registers on CPU. The over-usage of register will decrease the performance of GPU. Therefore, we store the packet data in global and texture memory and the filtering rules are stored in registers and constant memory.

Figure 6 illustrates the performance comparison between CPU-based and GPU-based BPF classification algorithms. The results show that the GPU-based BPF algorithm can achieve 8x ~10x speedup over CPU-based BPF algorithm for 65 million packets; especially by using Zero Copy technique. The 6 registers are held when global memory is utilized for computation. In this experiment, it needs 6 register to store the filter rules. Issuing a thread needs 12 registers. The total of the registers is 49,152 register (12 ×1,024 threads × 4 multi-processors). However, there are 32,768 registers on GTS450. It means that the data of extra 16,384 registers (524,288 bits) are transferred to local memory. Similarly, it needs 7 registers to store the filter rules to issue a thread by using texture memory, and the total of registers is 28,672. Therefore the performance of using global memory is impaired by using the registers exhaustively. It is obvious that the speedup of BPF by using global memory is slightly superior to that of using texture memory on register side as shown in Fig. 4.

B. *Perfromance Evaluation of BitMap-RFC with Various Memory Architectures*

In this experiment, we only implemented BitMap-RFC on CUDA with four memory architecture combinations shown in table 3. BitMap-RFC algorithm needs to build hash tables before filtering. Since the sizes of these tables are bigger than that of register files, the filter rules cannot be stored in registers. The results show that the GPU-based RFC algorithm can achieve 5x~7x speedup over CPU-based RFC algorithm in Fig. 7. Because CPU-based BitMap-RFC is much faster than CPU-based BPF, the performance enhancement is not dramatic to results in BPF.

Table 1. The revised filter rules for GPU-based packet classification

	Source (addr/mask)	Destination (addr/mask)	protocol	Prot number
Rule 1	140.128.0.0 /255.255.0.0	123.204.0.0 /255.255.0.0	*	*
Rule 2	*	*	TCP	HTTP
Rule 3	219.85.0.0 /255.255.0.0	123.204.0.0 /255.255.0.0	TCP	FTP

Table 2. Combination of various memory architectures for CUDA-BPF

	Packet Location	Rule Location	Data Transfer
1	Global	Constant	NA
2	Global	Constant	Zero Copy
3	Texture	Constant	NA
4	Texture	Constant	Zero Copy
5	Global	Register	NA
6	Global	Register	Zero Copy
7	Texture	Register	NA
8	Texture	Register	Zero Copy

Table 3. Combination of various memory architectures for CUDA-BitMap-RFC

	Packet Location	Rule Location	Data Transfer
1	Global	Constant	NA
2	Global	Constant	Zero Copy
3	Texture	Constant	NA
4	Texture	Constant	Zero Copy

C. *Perfromance Evaluation on part of CUDA device*

Figures 8 and 9 show the computation time of filtering, host-to-device transferring and device-to-host transferring which takes to correctly classify 65 million packets by BPF and BitMap-RFC, respectively. It is obvious that I/O operations take most of time in classifying packets. The I/O operation is dependent on the PCI bus. This cost can be reduced when the speed of PCI bus is enhanced. The performance of host-to-device transferring can be improved by ZeroCopy technique, buy the time in performing filter is longer. ZeroCopy allows threads to access main memory on host directly. Therefore, the executing time of filter action is sum of threads performing filter action and threads accessing main memory though PCI-e bus. It is reason that ZeroCopy take longer time to performing filter action than non-ZeroCopy.

From these experimental results, it is obvious that the performance of GPU-based packet classification is better than that of CPU-based packet classification. However, there

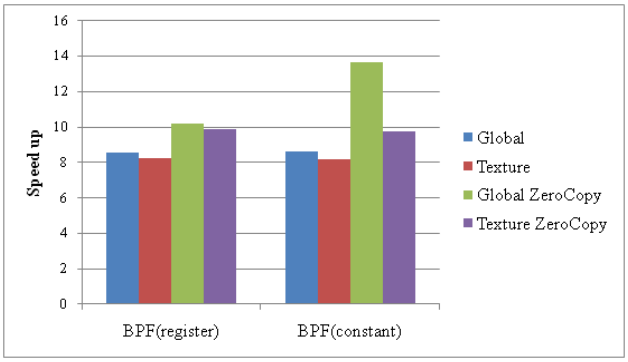


Figure 6. The performance comparision between sequentail BPF and GPU-based BPF. In GPU-based BPF, the packet data is stored in Global and Texture memory and the filter rules are stored in register and constant memory.

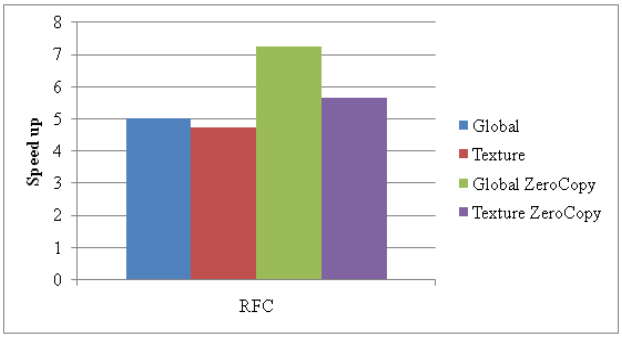


Figure 7. The performance comparision between CPU-based BitMap-RFC and GPU-based BitMap-RFC. In GPU-based BitMap-RFC, the packet data is stored in Global and Texture memory and the hash tables are stored in constant memory.

are two issues can be observed. First, the performances of BPF and BitMap-RFC with texture memory are worse than those with global memory. In the early CUDA 1.x device, global memory has no cache. Therefore texture memory has better computation performance. However, in the Fermi device the global memory already has L1 and L2 caches. In addition, the packet data should be transferred from global memory, and then the packet data is copied to texture memory. Secondly, the operation of branch instruction, such as *if-else*, is significantly more expensive than other operations in CUDA devices. In CUDA devices, the kernel executes all branch conditions, and then takes the result from the correct condition. CUDA-BPF is implemented by *if-else* instructions. Therefore, the performance of CUDA-BPF is worse than that of CUDA-BitMap-RFC as shown in Fig. 10.

V. CONCLUSTION

In this paper, we propose a parallel packet classification method by leveraging CUDA device. The proposed method includes two well-know filter algorithms, BPF and BitMap-RFC, to achieve rapid and reliable packet classification. In addition, we considered the performance of the various memory architectures available to CUDA kernels, and how such performance can be optimized. The experiments show the performance of several combinations of memory architectures and the detail of GPU computing time, such as filter computing, host-to-device and device-to-host transfer. Overall, our method can enhance the performance significantly.

In the future, we aim to develop a filter algorithm to reduce the times of accessing on-chip memory on GPU and decrease the usage of branch instruction. Also, we focus on improving our method to provide a real-time giga-bit network and fast network telescope packet analysis applications.

ACKNOWLEDGMENT

This research was partially supported by the National Science Council under the Grants NSC-99-2632-E-126-001-MY3.

REFERENCES

[1] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*,37(3):238-275, 2005.

[2] A. Nottingham, B. Irwin. Parallel packet classification using GPU co-processors. *SAICSIT Conf.ACM.*, 231-241, 2010.

[3] A. Nottingham, B. Irwin. GPU packet classification using OpenCL: a consideration of viable classification methods. *SAICSIT Conf.ACM.*, 160-169, 2010

[4] M. LCharalambous, P. Trancoso, A. Stamatakis. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor., *In Proceedings of the 10th Panhellenic Conference on Informatics*, LNCS, 415-425, 2005.

[5] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware., *Computer Graphics Forum*, 26:80-113, 2007.

[6] Nvidia cuda c best practices guide, version 4. Online, March 2011.

[7] A. Begel, S. McCanne, and S. L. Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. *SIGCOMM Comput. Commun. Rev.*, 29(4):123-134, 1999.

[8] S. Mccanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture., 259-269, 1993.

[9] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended tcams. *In ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, 120, 2003.

[10] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. *In SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 213-224, 2003.

[11] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191-202, 1998.

[12] D. R. Engler and M. F. Kaashoek. Dpf: Fast, flexible message demultiplexing using dynamic code generation. *In SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, 53-59, 1996.

[13] S. Mccanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. 259-269, 1993.

[14] F. Baboescu and G. Varghese. Scalable packet classification. *SIGCOMM Comput. Commun. Rev.*, 31(4):199-210, 2001

[15] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev*., 28(4):203-214, 1998.

[16] M. Yuhara, B. N. Bershad, C. Maeda, J. Eliot, and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages., *In Proceedings of the 1994 Winter USENIX Conference*, 153-165, 1994.

[17] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. Ffpf: Fairly fast packet filters., *In Proceedings of OSDI04*, 347-363, 2004.

[18] S. Ioannidis and K. G. Anagnostakis. Xpf: Packet filtering for low-cost network monitoring. *In Proceedings of the IEEE Workshop on High-Performance Switching and Routing* (*HPSR*), 121-126, 2002.

[19] Z. Wu, M. Xie, and H. Wang. Swift: a fast dynamic packet filter. *In NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 279-292, 2008.

[20] P. Gupta and N. McKeown, Packet Classification on Multiple Fields., *in Proc. of ACM SIGCOMM, Computation Communication Rev*., 29:147-160, 1999.

[21] T. Sherwood, G. Varghese and B. Calder, A Pipelined Memory Architecture for High Throughput Network Processors., *in Proc. of ACM ISCA'03*, 2003.

[22] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small Forwarding Tables for Fast Routing Lookups., *in Proc. of ACM SIGCOMM '97*, 3-14, 1997.

[23] W. Eatherton, G Varghese, and Z Dittia, Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates., *in Proc. of ACM SIGCOMM on Computer Communication Review*, 34(2): 97-122, 2004.

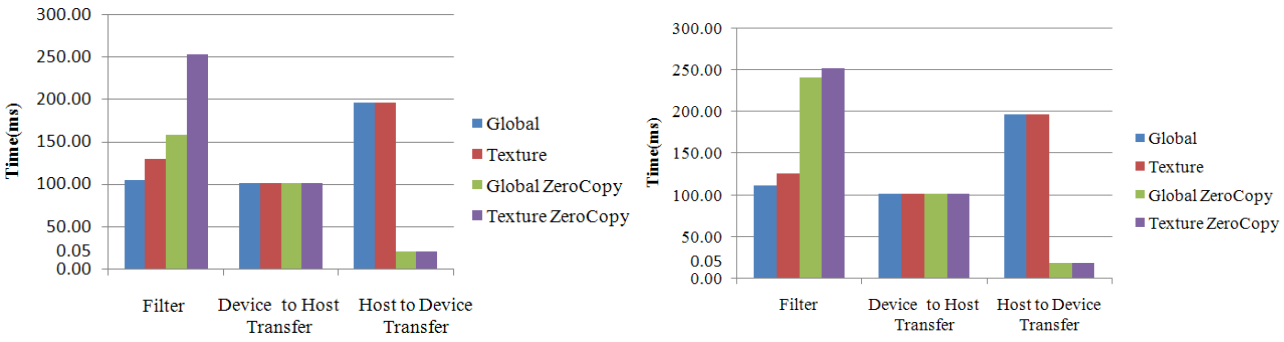
[24] Xianghui Hu, Xinan Tang, and Bei Hua, A High-performance IPv6 Forwarding Algorithm for a Multi-core and Multithreaded Network Processor., *in Proc. of ACM PPoPP'06*, 168-177, 2006.

[25] E. Spitznagel. Compressed Data Structures for Recursive Flow Classification., *Technical Report*, WUCSE-2003-65, 2003.

[26] D. Liu, B. Hua, X. Hu, and X. Tang, High-performance packet classification algorithm for many-core and multithreaded network processor., *in Proc. CASES*, 334-344, 2006.

[27] S. Han, K. Jang, K. Park and S. Moon, PacketShader: a GPU-accelerated Software Router., *In proceedings of ACM SIGCOMM 2010*.

[28] N.K. Govindaraju, S. Larsen, J. Gray and D. Manocha, "A memory model for scientific algorithms on graphics processors", *In SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006:89.



(a) Constant memory (b) Register

Figure 8. The executing time for parts of GPU-based BPF. (a) the filter rules are stored in Constant memory. (b) the filter rules are stored in Registers.

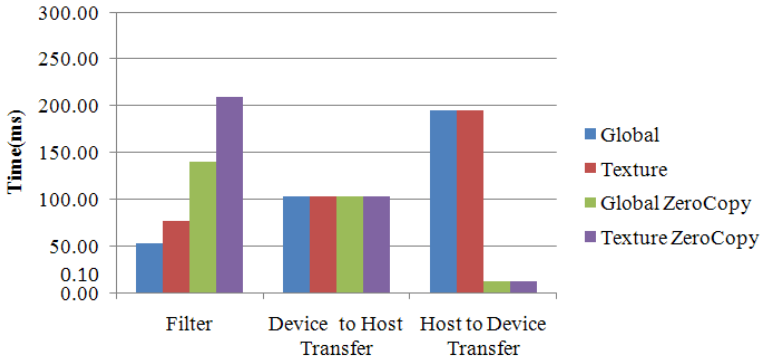


Figure 9. The executing time for parts of GPU-based BitMap-RFC. The filter rules are stored in Constant memory.

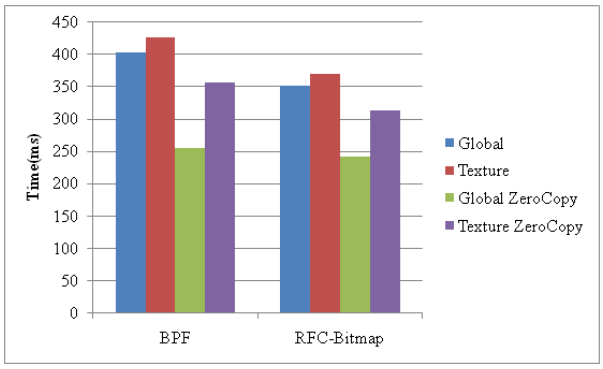


Figure 10. The performance comparision between GPU-based BPF and GPU-based BitMap-RFC.